CMPSC-265
Data Structures and Algorithms

Zaihan Yang
zyang13@suffolk.edu

Department of Math and Computer Science
Suffolk University

Fall 2019

Notice

- HW3 posted on Blackboard. Will be due on this Sunday midnight.

Recap

- Big O Notation: its definition and meanings
- Typical complexity classes in order
- $O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^3)<O(2^n)$
- How to represent the time complexity of a program code in terms of Big O notation;
  - Analyze the number of steps the codes will take (mostly by analyzing the inner loops) and represent it as a function g(n)
  - Discard all lower-order terms.
  - Ignore all constants
  - The resulting term is the f(n)
  - Represent g(n) = O(f(n))

Learning Topics

- Elementary sorting algorithms
- Implementing the Comparable and Comparator interface to facilitate sorting on objects.

Sorting

- Sorting is the process of rearranging a sequence of objects so as to put them in some logical order.
  - To sort a list of numbers.
  - Name of students in alphabetical order
  - Restaurants in order of rating
- Sorting is everywhere and important:
  - In early days of computing, up to 30% of all computing cycles was spent on sorting.
  - Plays major role in commercial data processing and in modern scientific computing, and many applications.
  - Often the first step to organizing data, and is often the starting point to solve other problems.
    - Can Make search easier and more efficient

How to Sort?

- We see the entire list at once but the computer does not

- Computer needs to compare two elements and do swap/ move
  - We need to tell the computer how to compare and move elements step by step (sorting algorithm)

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 78

Sorting Algorithm

- We will learn a bunch of sorting algorithms. Today for the elementary ones.
  - The general idea (algorithm)
  - How to implement the algorithm in codes
  - Analyze its time complexity for the best case, worst case and average case in Big O Notation
  - Analyze its memory usage:
    - in-place: use no extra memory or extra memory not proportional to input size;
    - extra memory needed: i.e. to hold another copy of the array.

Elementary Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort

Selection Sort

- One of the simplest sorting algorithms;
- Basic idea:
  - Iterative process over a given array *a*;
  - In each iteration i^{th} $(0<=i<a.length-1)$, find the i^{th} smallest element in *a*, and exchange it with the i^{th} entry.
- It works by repeatedly selecting the smallest remaining items, and therefore gets its name.

Selection Sort: example

#1 iteration (i=0)

i	<---------------	j	------------------>	
2	6	7	3	1
1	6	7	3	2

#2 iteration (i=1)

	i	<----------	j	--------------->
1	6	7	3	2
1	2	7	3	6

#3 iteration (i=2)

1	2	7	3	6
1	2	3	7	6

#4 iteration (i=3)

1	2	3	7	6
1	2	3	6	7

#5 iteration (i=4)

1	2	3	6	7
1	2	3	6	7

Final result

Selection Sort Implementation

```
public void selectionSort(int arr[])
  {
      int n = arr.length;
      for (int i = 0; i < n-1; i++)
      {
          // Find the minimum
          int minIndex = i;
          for (int j = i+1; j < n; j++)
              if (arr[j] < arr[minIndex])
                  minIndex = j;

          // Swap
          int temp = arr[minIndex];
          arr[minIndex] = arr[i];
          arr[i] = temp;
      }
  }
```

Selection Sort: performance analysis

- How many passes through the array?

- In each pass, How many comparisons and swaps?

Selection sort: performance analysis

- How many passes: N-1 passes
- How many comparisons need to make?
- Two nested for loops:
- outer loop: i from 0 to n-1; (n=a.length):

  n-1 steps;
- inner loop: for each i, j from i+1 to n

  n-i-1 steps
- So: (n-1)+(n-2)+(n-3)+……2+1+0 steps:

  the sum of this sequence is: $n(n-1)/2 \sim n^2/2$

  (quadratic)

Selection sort: performance analysis

- How many exchanges need to make?
  - for each iteration i: 0<1<n, at most make one exchange.
  - so: at most n exchanges ~ n (linear): good
- The total number of operations:
  - **$\sim (n^2/2+n) \sim n^2$**
- Running time is insensitive to the initial input (not that good):
  - no matter the array is already in-sort.
  - or is randomly sorted.

Selection sort: performance analysis

N-N matrix:

Comparisons:

Unshaded entries correspond to compares; One-half of entries in the table are unshaded. On and above the diagonal.

$\sim n^2/2$

Exchanges:

The entries on the diagonal:

$\sim n$

Total: $n^2/2+n \sim n^2$

a[]

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

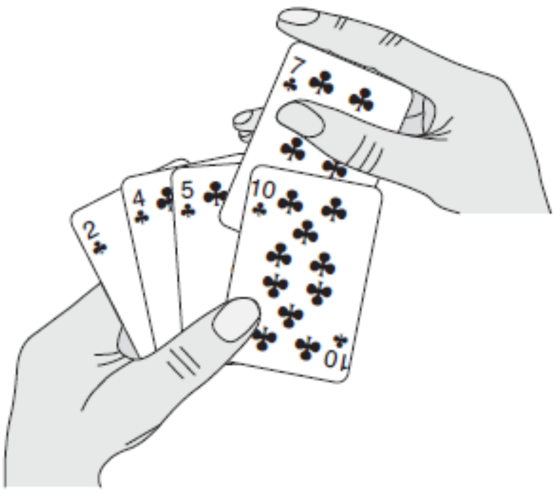
entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

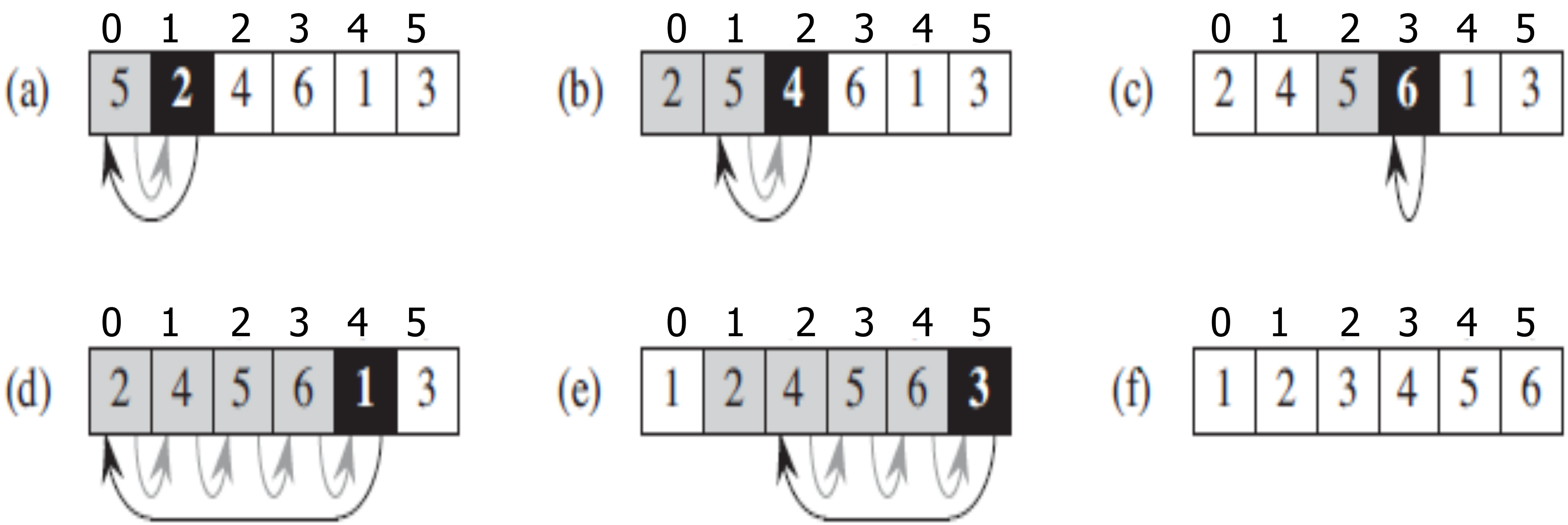
Insertion Sort

- Basic idea:
- Borrows the idea from people playing bridge hands:
  - People often insert each card into its proper place among those already sorted.



- Computer implementation:
  - Make space to insert the current item by moving larger items one position to the right, before inserting the current item into the vacated position.

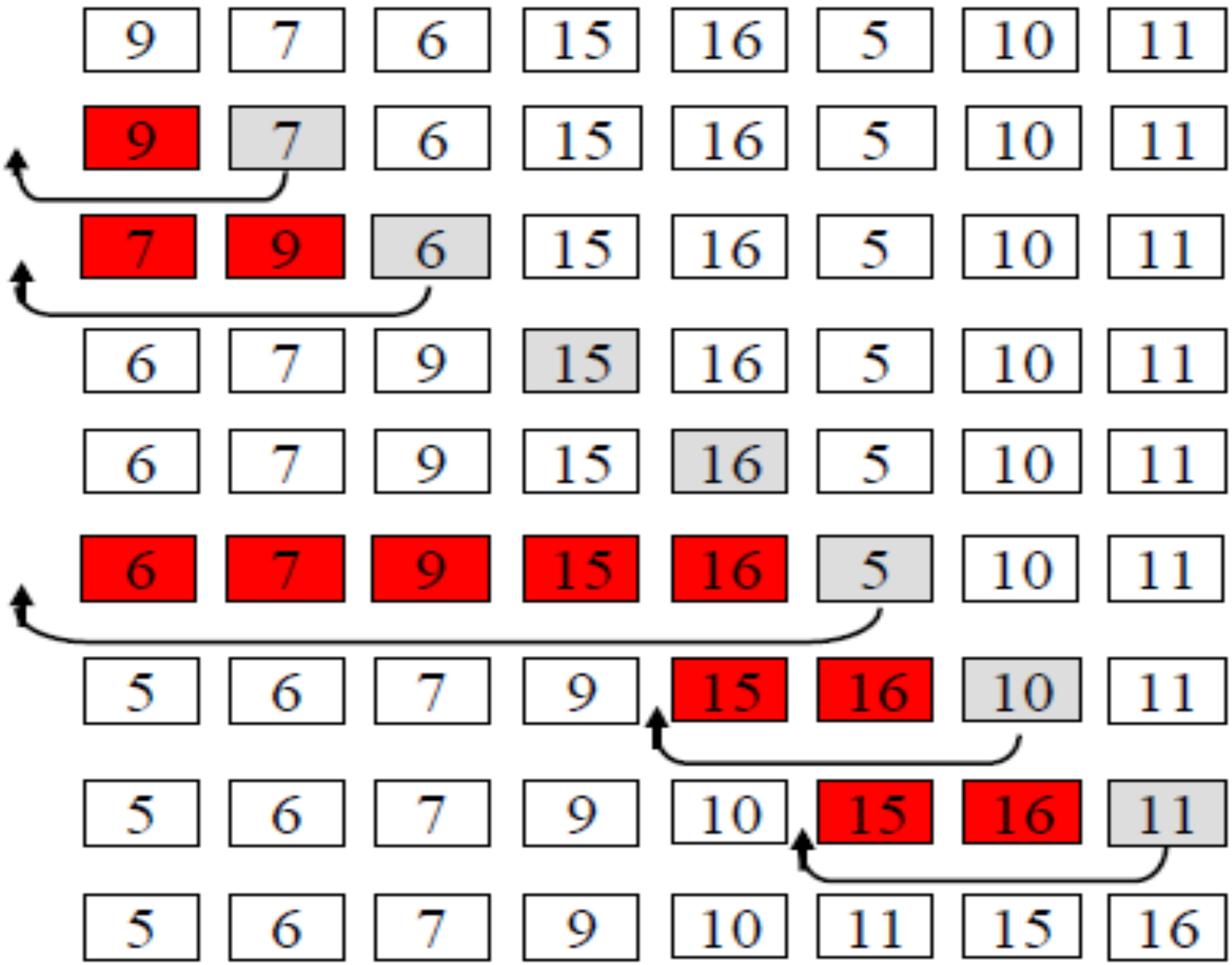
Insertion Sort: examples



Insertion Sort: practice

9	7	6	15	16	5	10	11

Insertion Sort: practice



Insertion Sort Implementation

```
void insertionSort(int arr[])
  {
     int n = arr.length;
     for (int i=1; i<n; i++)
     {
        int key = arr[i];
        int j = i-1;

        while (j>=0 && arr[j] > key)
        {
           arr[j+1] = arr[j];
           j = j-1;
        }
        arr[j+1] = key;
     }
  }
```

Insertion Sort: performance analysis

i	j	a[] 0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

Comparisons and Exchanges: to count the number of entries below the diagonal

Insertion Sort: performance analysis

- Worst case: the original array is in a reversed order.
  - Comparisons: all the entries below the diagonal: $\sim n^2/2$
  - Exchanges == Comparisons: $\sim n^2/2$
  - Total: $\sim n^2$ $O(n^2)$
- Best case:
  - Comparisons: n (1 comparison in each iteration) $\sim n$
  - Exchanges: 0
  - Total: $\sim n$ $O(n)$
- Average Case: assume each item will go about halfway back.
  - Comparisons: $\sim n^2/4$
  - Exchanges: $\sim n^2/4$
  - Total: $\sim n^2$ $O(n^2)$

Elementary Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort

Bubble Sort Idea

- Compare neighbors and swap if not in order. Repeat until everything is in order.

- What do I get after one pass?

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 80

Bubble Sort: Example

- First Pass:
- (5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
- (1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4
- (1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
- (1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

- Second Pass:
- (1 4 2 5 8) –> (1 4 2 5 8)
- (1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2
- (1 2 4 5 8) –> (1 2 4 5 8)
- (1 2 4 5 8) –> (1 2 4 5 8)
- Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Bubble Sort: Example

- Third Pass:
- (1 2 4 5 8) –> (1 2 4 5 8)
- (1 2 4 5 8) –> (1 2 4 5 8)
- (1 2 4 5 8) –> (1 2 4 5 8)
- (1 2 4 5 8) –> (1 2 4 5 8)

Bubble Sort Implementation

```
public void bubbleSort(int arr[])
  {
      int n = arr.length;
      for (int i = 0; i < n-1; i++)
          for (int j = 0; j < n-i-1; j++)  // why here is j<n-i-1?
              if (arr[j] > arr[j+1])
              {
                  int temp = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = temp;
              }
  }
```

What if the input array is already sorted?

Use a boolean flag

Bubble Sort Better Implementation

```
public void bubbleSort(int arr[])
{
        int n = arr.length;
        boolean swapped;
        for (int i = 0; i < n - 1; i++)
        {
            swapped = false;
            for (int j = 0; j < n - i - 1; j++)
            {
                if (arr[j] > arr[j + 1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            if (swapped == false)
                break;
        }
    }
```

Complexity of Bubble Sort

- How many passes through the array?
- How many comparisons and swaps?
- Complexity order: $O(n^2)$